

**Rec : T e M**

*(Wa & M - Cha e 2)*

*T e a e h a , e c e, d e.*

- L. Peter Deutsch

*I ee e e b rather ha d  
de a d!*

- Lewis Carroll

- A *recursive function* is a function that calls itself.
- Anything that can be solved *by a recursive function* can be solved *by an iterative function* and vice versa.
- Sometimes a recursive solution can be expressed more *clearly* and *concisely* than an iterative one.

## *fac* a Function ( ! )

$$\mathit{factorial}(0) = 1 \text{ (by definition)} = 1$$

$$\mathit{factorial}(1) = 1*1 = 1*\mathit{factorial}(0)$$

$$\mathit{factorial}(2) = 2*1 = 2*\mathit{factorial}(1)$$

$$\mathit{factorial}(3) = 3*2*1 = 3*\mathit{factorial}(2)$$

$$\mathit{factorial}(4) = 4*3*2*1 = 4*\mathit{factorial}(3)$$

$$\mathit{factorial}(5) = 5*4*3*2*1 = 5*\mathit{factorial}(4)$$

$$\mathit{factorial}(6) = 6*5*4*3*2*1 = 6*\mathit{factorial}(5)$$

# Recursive Definition of $fac$ $a ( )$

$$fac \ a ( ) = \begin{cases} 1 & f = 0 \\ * \ fac \ a ( -1 ) & f > 0 \end{cases}$$

- How would we implement this in C++ ?

<b>Function Definition:</b>	<b>C++ Implementation:</b>
$\begin{aligned} \text{factorial}(n) = \\ 1 \qquad \qquad \qquad \text{if } n = 0 \\ n * \text{factorial}(n-1) \text{ if } n > 0 \end{aligned}$	<pre>int factorial( n ) {     if( n == 0 )         return 1;     else         return n*factorial( n-1 ); }</pre>

# Understanding Recursion

- You can think of a recursive function call as if it were calling a completely separate function.
- In fact, the *e a* that can be performed by both functions is the same, but the *da a* input to each is different

# Understanding Recursion (Cont'd.)

```
int factorialA( int n )
{
    if( n == 0 )
        return 1;
    else
        return n*factorialB( n-1 );
}
```

```
int factorialB( int m )
{
    if( m == 0 )
        return 1;
    else
        return m*factorialC(m-1);
}
```

- If factorialB( ) and factorialC( ) perform the same operations as factorialA( ), then factorialA( ) can be used in place of them.

## Example: *fac*    *a* (3)

factorial(3):  $n = 3$  calls factorial(2)

factorial(2):  $n = 2$  calls factorial(1)

factorial(1):  $n = 1$  calls factorial(0)

factorial(0): returns 1 to factorial(1)

factorial(1):  $1 * \text{factorial}(0)$  becomes  $1 * 1 = 1$

: returns 1 to factorial(2)

factorial(2):  $2 * \text{factorial}(1)$  becomes  $2 * 1 = 2$

: returns 2 to factorial(3)

factorial(3):  $3 * \text{factorial}(2)$  becomes  $3 * 2 = 6$

: returns 6



Example:  $S \ c \ g \ Sa \ age$

- **P b e** : Slice a sausage from back to front.  
(Assume that sausages have distinguishable front and back ends.)
- **S a e** : Slicing a sausage into  $N$  slices from back to front can be decomposed into  $a \ g \ a \ g \ e \ ce \ a \ he \ e \ d$  (which is “easy”) and  $a \ g \ he \ e \ a \ g \ N-1 \ ce$  from back to front (which is a smaller problem of the “same type”).

## Slicing Sausage (Cont'd)

- **Proof** : If we keep reducing the length of the sausage to be sliced, we will eventually end up with 1 slice left.
  - We could even go a step further and end with a sausage of length 0, which requires no slicing.
- **Theorem** : Since our strategy reduces the size of the sausage by 1 slice each step, we will eventually reach the *base case* (0 slices left).

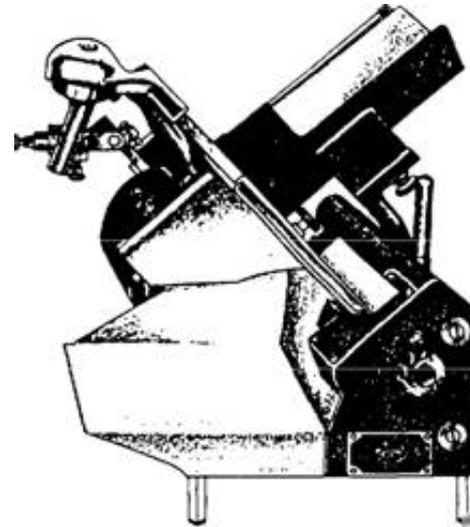
*L e ! He e he a ...*



0

1

2



Butcher #2 *a a a e hef ce* at the rightmost end. He then passes the sausage to butcher #1, who makes the next cut, followed by butcher #0. They take turns with the only sausage slicer in their shop.

# Sausage Slicer ( C++ )

```
#define make1slice    cout

void sausageSlicer( char sausage[], int size )
{
    if( size > 0 )
    {
        // slice the end off
        make1slice << sausage[ size-1 ];
        // slice the rest of the sausage
        sausageSlicer( sausage, size-1 );
    }
    // base case: do nothing if size == 0
}
```

# Trial Run

- Suppose `char e e []` contains `F, D, A`
- Executing

```
sausageSlicer( pepperoni, 3 );
```

results in

sausage 

F	D	A
---	---	---

size 

3
---

## Trial Run (Cont'd.)

- Since  $\text{size} = 3 > 0$ ,  
    `make1slice << sausage[ size-1];`  
will cause `sausage[2]`, containing 'A', to be sliced off.
- After this  
    `sausageSlicer( sausage, 2 );`  
is executed.

## Trial Run (Cont'd.)

- Executing

```
sausageSlicer( sausage, 2 );
```

causes

```
make1slice << sausage[size-1];
```

to be executed, which results in `sausage[1]`, containing 'D', to be sliced off.

- After this

```
sausageSlicer( sausage, 1 );
```

is executed.

## Trial Run (Cont'd.)

- Executing

```
sausageSlicer( sausage, 1 );
```

causes

```
make1slice << sausage[size-1];
```

to be executed, which results in `sausage[0]`, containing 'F', to be sliced off.

- After this

```
sausageSlicer( sausage, 0 );
```

is executed.

## Trial Run (Cont'd.)

- Executing

sausageSlicer( sausage, 0 );

does  $h$   $g$  and returns to the place where it was called.

## Trial Run - *Re Pa h*

- `sausageSlicer( sausage, 0 )` returns to `sausageSlicer( sausage, 1 )`, which has nothing left to do.
- `sausageSlicer( sausage, 1 )` returns to `sausageSlicer( sausage, 2 )`, which has nothing left to do.
- `sausageSlicer( sausage, 2 )` returns to `sausageSlicer( sausage, 3 )`, which has nothing left to do.
- `sausageSlicer( sausage, 3 )` returns to `sausageSlicer( pepperoni, 3 )`, the original call to `sausageSlicer( )`, and execution is done.

## Trial Run - $Ke P$

Note that there is *only one* *algorithm*, (i.e. one recursive function), but it is used over and over on successively smaller pieces of the original sausage until, finally, the entire sausage is sliced.

## New Strategy for a $N \times T$

- S a e** : Slicing a sausage into  $N$  slices from back to front can be decomposed into  $c g$   $N-1$   $ce f$   $bac$   $f$  (a smaller problem of the same type) and  $a g a g e ce a he f$  (which is “easy”).
- P e & T e a** : Since, as before, our strategy reduces the size of the sausage by 1 slice each step, we will eventually reach  $ca$  (0 slices left).

*Ne T Ne S a eg*



0

1

2



This time, someone hands the sausage to butcher #0. As the senior member of the team, he will slice only if the others have done their work. So, he passes the sausage to butcher #1 who, in turn, passes the sausage to butcher #2. Butcher #2 makes the first slice, as before, at the rightmost end of the sausage, and then passes it back to the other two butchers, who can now complete their tasks.

# *Ne* Sausage Slicer in C++

```
int size;          // global variable containing size of sausage

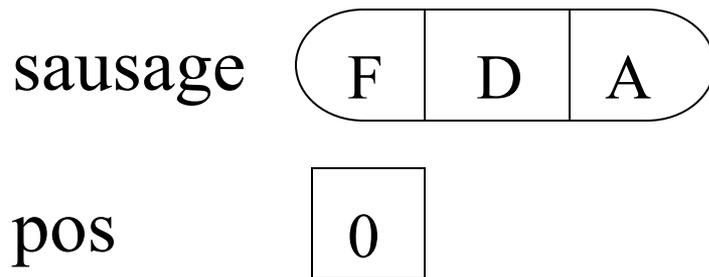
void sliceAsausage( char sausage[], int pos )
{
    if( pos < size )
    { // cut into slices everything to the right of sausage[ pos ]
        sliceAsausage( sausage, pos+1 );
        // slice off sausage[ pos ];
        make1slice << sausage[ pos ];
    }
    // base case: do nothing if pos == size (i.e. past end of sausage)
}
```

# Trial Run of New Sausage Slicer

- Suppose, as before, `char e[]` contains `F, D, A` and `ed` is initialized to 3.
- Executing

```
sliceAsausage( pepperoni, 0 );
```

results in



## New Slicer Trial Run (Cont'd.)

- Since  $\text{pos} = 0 < \text{size}$ ,  
    `sliceAsausage( sausage, 1 );`  
will be executed.
- After this  
    `sliceAsausage( sausage, 2 );`  
is executed, followed by  
    `sliceAsausage( sausage, 3 );`

# New Slicer Trial Run - Return Path

- `sliceAsausage( sausage, 3 )` does nothing since `pos = size`.
- `sliceAsausage( sausage, 3 )` returns to `sliceAsausage( sausage, 2 )`, which prints `sausage[2] = 'A'`.
- `sliceAsausage( sausage, 2 )` returns to `sliceAsausage( sausage, 1 )`, which prints `sausage[1] = 'D'`.
- `sliceAsausage( sausage, 1 )` returns to `sliceAsausage( sausage, 0 )`, which prints `sausage[0] = 'F'`.
- `sliceAsausage( sausage, 0 )` returns to `sliceAsausage( pepperoni, 0 )`, and execution is done.

*The e e ha e a  
ce a a age!*

# $X^n$ Function

$$X^n = 1 \quad \text{if } n = 0 \text{ (base case)}$$

$$X^n = X * X^{(n-1)} \quad \text{if } n > 0$$

This can easily be translated into C++. However, a *efficient* definition is possible:

$$X^n = 1 \quad \text{if } n = 0 \text{ (base case)}$$

$$X^n = [X^{(n/2)}]^2 \quad \text{if } n > 0 \text{ and even}$$

$$X^n = X * [X^{(n-1)/2}]^2 \quad \text{if } n > 0 \text{ and odd}$$



# Fibonacci Sequence

The first two terms of the sequence are 1, and each succeeding term is the sum of the previous pair.

$$1 \quad 1$$

$$1 + 1 = 2$$

$$1 + 2 = 3$$

$$2 + 3 = 5$$

$$3 + 5 = 8$$

$$5 + 8 = 13 \dots, \text{ or}$$

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \quad 233 \quad 377 \quad 610 \dots$$

# Fibonacci Sequence (Cont'd.)

F D :	C++ I a :
$\text{fib}(1) = 1 \quad (\text{base case})$ $\text{fib}(2) = 1 \quad (\text{base case})$ $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2),$ <p style="text-align: right;"><math>\text{for } n &gt; 2</math></p>	<pre>int fib( int n ) {     if( n &lt;= 2 )         return 1;     else         return fib(n-1) + fib(n-2); }</pre>

- Base case added in the recursive function

- Very inefficient: fib(7) will call fib(3) five times!

# Fibonacci Sequence with Rabbits

- Problem posed by Fibonacci in 1202:
  - A pair of rabbits 1 month old are too young to reproduce.
  - Suppose that in their 2<sup>nd</sup> month and every month thereafter they produce a new pair.
  - If each new pair of rabbits does the same, and none of them die, how many pairs of rabbits will there be at the beginning of each month?

# Fibonacci Sequence with Rabbits (Cont'd.)

Month 1: # Pairs: 1 Adam & Eve  
2: 1 Adam & Eve  
3: 2 Adam & Eve have twins1  
4: 3 Adam & Eve have twins2  
5: 5 Adam & Eve have twins3;  
twins1 have twins4  
6: 8 Adam & Eve have twins5;  
twins1 have twins6; twins2 have twins7

- *Recurrence relation:  $f_n = f_{n-1} + f_{n-2}$*

# Fibonacci Sequence - Other Applications

- A male bee has only one parent (his mother), while a female bee has a father and a mother. The number of ancestors, per generation, of a male bee follows the Fibonacci sequence.
- The number of petals of many flowers are Fibonacci numbers.
- The number of leaves at a given height off the ground of many plants are Fibonacci numbers.

# Mad Scientist's Problem

A mad scientist wants to make a straight chain of length  $n$  out of pieces of lead and plutonium. However, the mad scientist is *dumb*! He knows that if he puts two pieces of plutonium next to each other, the whole chain will explode. How many safe, linear chains are there?

Example:  $n = 3$

L L L (safe)

P L L (safe)

L L P (safe)

P L P (safe)

L P L (safe)

P P L (unsafe)

L P P (unsafe)

P P P (unsafe)

Result: 5 safe chains

# Mad Scientist (Cont'd.)

Let  $C(n)$  = number of safe chains of length  $n$

$L(n)$  = number of safe chains of length  $n$  ending with lead

$P(n)$  = number of chains of length  $n$  ending with plutonium

Now, the total number of safe chains of length  $n$  must be the sum of those that end with lead and those that end with plutonium, namely

$$C(n) = L(n) + P(n)$$

## Mad Scientist (Cont'd.)

Note that we make a chain of length  $n$  by adding to a chain of length  $n-1$ .

So, consider all chains of length  $n-1$ . Note that we can add a piece of lead to the end of each of these, since this will not make the chain unsafe.

Therefore,

$$L(n) = C(n-1)$$

## Mad Scientist (Cont'd.)

Consider again all chains of length  $n-1$ . Note that we can add a piece of plutonium to the end of only the chains that end with lead.

Therefore,

$$P(n) = L(n-1)$$

# Mad Scientist (Cont'd.)

Substituting formulas for  $L(n)$  and  $P(n)$  in the formula for  $C(n)$  we see that

$$\begin{aligned}C(n) &= L(n) + P(n) \\ &= C(n-1) + L(n-1) \\ &= C(n-1) + C(n-2), \quad \text{since } L(k) = C(k-1) \text{ for any } k\end{aligned}$$

Note that this is the Fibonacci recursion!

However, the base case(s) are different:

$$C(1) = 2 \quad L \text{ or } P$$

$$C(2) = 3 \quad LL \text{ or } LP \text{ or } PL$$

## Mad Scientist (Cont'd.)

Back to our example with  $n = 3$ :

$$\begin{aligned}C(3) &= C(2) + C(1) \\ &= 3 + 2 \\ &= 5\end{aligned}$$

which agrees with the answer we found by enumerating all the possibilities.

# Mr. Spock's Dilemma

There are  $n$  planets in an unexplored planetary system, but there is only time (or fuel) for  $k$  visits.

How many ways are there for choosing a group of planets to visit?

Let  $C(n, k)$  denote the number of ways to choose  $k$  planets from among  $n$  candidates.

# Mr. Spock's Dilemma: $S$ $S$ $a$ $e$ $g$

Consider planet Vega. Either we visit Vega or we don't.

- *If*  $e$   $S$  Vega, then we will have to choose  $k-1$  other planets to visit from the remaining  $n-1$ .
- *If*  $e$   $d$   $S$  Vega, then we will have to choose  $k$  other planets to visit from the remaining  $n-1$ .
- Therefore,

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad \text{for } 0 < k < n$$

# Mr. Spock's Dilemma: *Rec* *C e a*

Consider the criteria for constructing a recursive solution:

1) **S a e** : Is the original problem defined in terms of smaller problems of the same type? *Ye ,*

$$C( , ) = C( -1, -1 ) + C( -1, )$$

2) **P e** : Does each recursive call diminish the size of the problem? *Ye , f a g e f C dec ea e h*

*each ec e ca a d ec d a g e d e*  
*c ea e.*

3) **Te a** : Will a "base case" be reached eventually?

*Le ee ha ba e ca e a e eeded, a d he ee f e*  
*f he a a be eached.*

# Mr. Spock's Dilemma: *Base Case*

- Note that the recursion formula

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

only applies when  $0 < k < n$ . Consequently, we need to consider  $k < 0$ ,  $k = 0$ ,  $k = n$ , and  $k > n$ .

- Since there is only 1 way to choose 0 planets and only 1 way to choose all  $n$  planets, we have

$$C(n, k) = 1 \text{ if } k = 0 \text{ or } k = n$$

- Since it is not possible to choose  $< 0$  planets or  $> n$  planets,

$$C(n, k) = 0 \text{ if } k < 0 \text{ or } k > n$$

$$B a e C a e (C d.)$$

- Putting this all together, we have

$$C(n, k) =$$

$$0 \quad \text{if } k < 0 \text{ or } k > n \quad (ba e ca e)$$

$$1 \quad \text{if } k = 0 \text{ or } k = n \quad (ba e ca e)$$

$$C(n-1, k-1) + C(n-1, k) \quad \text{if } 0 < k < n$$

- Consider the recursion formula, where  $0 < k < n$ . Since the first argument of  $C(n, k)$  decreases with each recursive call and second argument does not increase, eventually either  $n = k$  or  $k = 0$ . Both  $ba e ca e$  are defined above. Therefore,  $e a a ed$ .

# Mr. Spock's Dilemma: $S$

$C++$

```
int C( int n, int k )    // # of ways to choose k of n things
{
    if( k == 0 || k == n ) return 1;
    if( k < 0 || k > n ) return 0;

    return C( n-1, k-1 ) + C( n-1, k );
}
```

# Binary Search: *Te e h e B*

- Problem: Search the telephone book for someone's phone number.
- Binary Search Strategy:
  - a) Open the book somewhere near the middle.
  - b) If the the person's name is in the first half, ignore the second half, and search the first half, starting again at step a).
  - c) If the the person's name is in the second half, ignore the first half, and search the second half, starting again at step a).
  - d) If the person's name is on a given page, scan the page for the person's name, and find the phone number associated with it.

# Binary Search: *Search a A a*

- Problem: Given an array,  $A[ ]$ , of  $n$  integers, sorted from smallest to largest, determine whether value  $v$  is in the array.
- Binary Search Strategy:
  - If  $n = 1$  then check whether  $A[0] = v$ . Done.
  - Otherwise, find the midpoint of  $A[ ]$ .
  - If  $v > A[\text{midpoint}]$  then recursively search the second half of  $A[ ]$ .
  - If  $v \leq A[\text{midpoint}]$  then recursively search the first half of  $A[ ]$ .

## *Search A a* : C++ Implementation

```
int binarySearch( int A[ ], int v, int first, int last )
{
    if( first > last ) return -1;           // v not found in A[ ]
    int mid = (first + last)/2;           // set mid to midpoint
    if( v == A[mid] ) return mid;
    if( v < A[mid] ) return binarySearch( A, v, first, mid-1 );
    return binarySearch( A, v, mid+1, last );
}
```

# C++ Implementation (Cont'd.)

Two common mistakes:

1) CORRECT: `mid = ( first + last )/2;`

INCORRECT: `mid = ( A[first] + A[last] )/2;`

2) CORRECT: `return binarySearch( A, v, mid+1, last );`

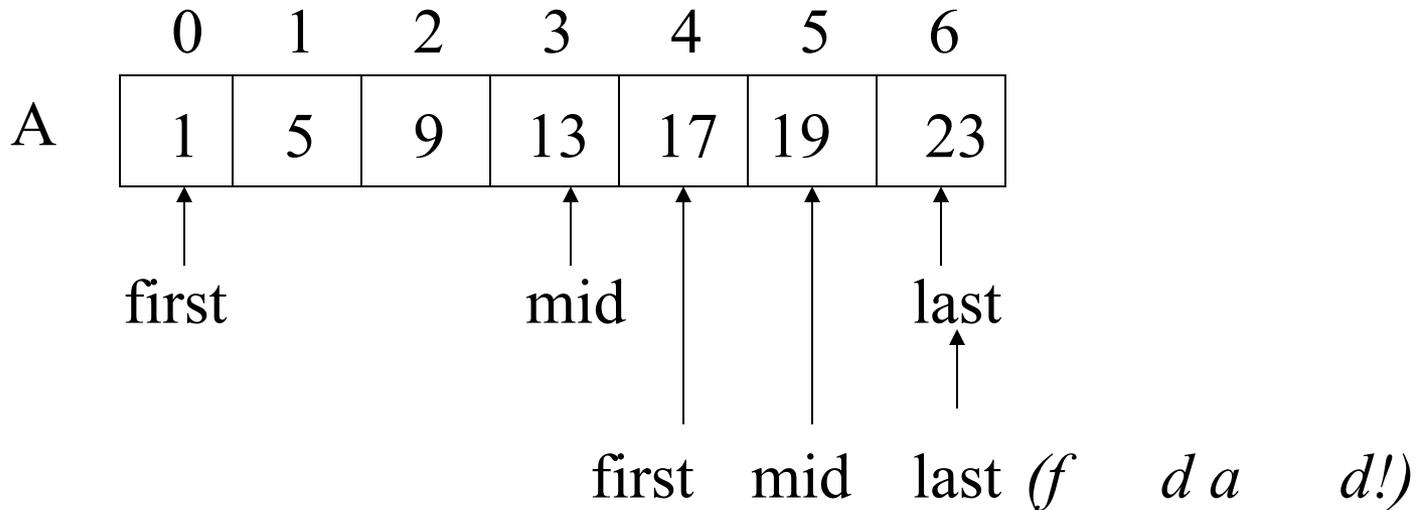
INCORRECT: `return binarySearch( A, v, mid, last );`

## *Search A a* : Implementation Notes

- The whole array,  $A[ ]$ , is passed with each call to `binarySearch( )`.
- The active part of array  $A[ ]$  is defined by  $f$  and  $a$  .
- A return value of -1 means that  $a$  was not found.

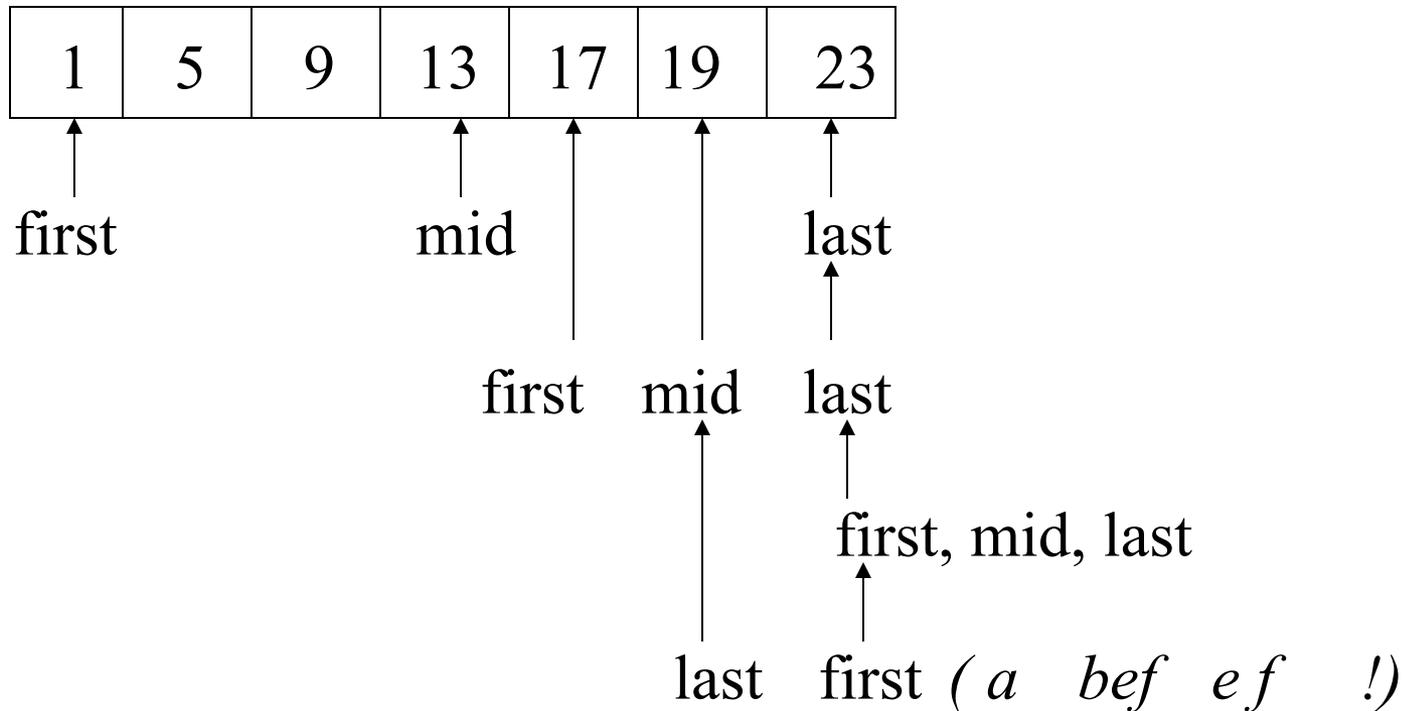
# Search in A : Example

- Suppose  $A[]$  contains  $\{1, 5, 9, 13, 17, 19, 23\}$ , and we are interested in searching for **19**.
- Executing  $\text{Search}(A, 19, 0, 6)$ ; results in



# Search in $A$ : Example (Cont'd.)

- Suppose we are interested in searching for **21**:



## *Search Algorithms* : Final Comments

- Suppose that we have an array of a million numbers.
- The first decision of a binary search will eliminate approximately half of them, or 500,000 numbers.
- The second decision will eliminate another 250,000.
- Only 20 decisions are needed to determine whether a given number is among a sorted list of 1 million numbers!
- A *linear* search might have to examine *a fraction* .
- Additional Note: Binary searching through a *b* numbers would require about 30 decisions, and a *a* numbers would (theoretically) require only 40 decisions.